Article
Sergei Sarkisian · Jul 18 12m read

Angular. Container and presentational components (aka smart-dumb)

Hi! Today I would like to talk about one of the most important architectural patterns in Angular.

The pattern itself is not related to Angular directly, but as Angular is component-driven framework, this pattern is one of the most essential for building modern Angular applications.

Container-Presentation pattern

It is believed that good components should be small, focused, independent, testable and most important - reusable.

If your component is making server calls, contains business logic, tightly coupled to other components, knows too much about other component's or services' internals, then it become bigger, harder to test, harder to extend, harder to reuse and harder to modify. To solve these problems the pattern "Container-Presentation" exists.

Generally, all the components can be divided in two groups: Container (smart) and Presentational (dumb) components.

Container components can get data from services (but should not call server APIs directly), contain some business-logic and serve data to services or children components. Usually, container components are those we specify as routed components in our routing configuration (but not always, of course).

Presentational components must only take data as input and display it on the screen in some way. They can react on user inputs, but only by changing its local isolated state. All the communications with the rest of the app should be done by emitting custom events. These components should be highly reusable.

For better context I will name some examples of container and presentational components:

Container: AboutPage, UserPage, AdminPanel, OrderPage, etc.

Presentational: Button, Calendar, Table, ModalDialog, TabView, etc.

Example of a crazy button

Let's look at extremely bad example of misusing the components approach I faced myself in one of the real projects.

```
@Component({
  selector: 'app-button',
  template: `<button class="className" (click)=onClick()>{{label}}</button>`
})
export class ButtonComponent {
  @Input() action = '';
  @Input() className = '';
```

```
  @Input() label = '';

  constructor(
    private router: Router,
    private orderService: OrderService,
    private scanService: ScanService,
    private userService: UserService
  ) {}

  onClick() {
    if (this.action === 'registerUser') {
      const userFormData = this.userService.form.value;
      // some validation of user data
      // ...
      this.userService.registerUser(userFormData);
    } else if (this.action === 'scanDocument') {
      this.scanService.scanDocuments();
    } else if (this.action === 'placeOrder') {
      const orderForm = this.orderService.form.values;
      // some validation and business logic related to order form
      // ...
      this.orderService.placeOrder(orderForm);
    } else if (this.action === 'gotoUserAccount') {
      this.router.navigate('user-account');
    } // else if ...
  }
}
```

I simplified it for better readability, but in reality things was much worse. This is a button component which contains all of the possible actions user can invoke by clicking button – making API calls, validating forms, retrieving information from the services and more. You can imagine how fast such component can become a hell in even relatively small application. The code of such button component I found (and then refactored) was more than 2000 lines long. Insane!

When I asked the developer who wrote it, why he decided to put all of this logic in the single component, he said that this is "encapsulation"

Let's remember what qualities good components should have:

Small - this button with 2000+ lines of code isn't small. And it will become bigger every time someone will need another button for different action.

Focused - this button do a lot of absolutely unrelated things and can't be called focused.

Independent - this button is tightly coupled with several services and forms, and changing any of them will affect the button.

Testable - no comments.

Reusable - it is not reusable at all. You need to modify component's code every time you want to use it for action it doesn't have, and you will get all the unneeded actions and dependencies this button has.

Moreover, this button component hides native HTML button under the hood, blocking access to its properties for developer. This is a good example of how component should not be written.

Let's put some very simple example of component which uses this button component and then will try to refactor them with Container-Presentation pattern.

```
@Component({
  selector: 'app-registration-form',
  template: `<form [formGroup]="userService.form">
  <input type="text" [formControl]="userService.form.get('username')" placeholder="Ni
ckname">
  <input type="password" [formControl]="userService.form.get('password')" placeholder
="Password">
  <input type="password" [formControl]="userService.form.get('passwordConfirm')" plac
eholder="Confirm password">
  <app-button className="button accent" label="Register" action="registerUser"></app-
button>
</form>
`
})
export class RegistrationFormComponent {
  constructor(public userService: UserService) {}
}
```

You can see that there is no logic inside this component – the form stored in a service, and the button contains all the logic invoking by click on it. So our button now have all the logic unrelated to its behavior, and smarter than its parent component which is directly related to the actions processed by button.

Refactor button component with Container-Presentation pattern

Let's separate the functions of these two components. Button should be presentational component – small and reusable. Registration form which contains the button can be container component which will hold business logic and communications with services layer.

We will not cover the part with unhiding the native button (I will probably cover this in some future article), but will focus mainly on architectural aspect of relation between these two components.

Refactored button component (presentational)

```
@Component({
  selector: 'app-button',
  template: `<button class="className" (click)=onClick()>{{label}}</button>`
})
export class ButtonComponent {
  @Input() className = '';
  @Input() label = '';

  @Output() click: EventEmitter = new EventEmitter();

  onClick() {
     this.click.emit();
  }
}
```

As you see, our refactored button is very simple. It has only two Inputs and one Output. Inputs are used for taking data from parent component and display it to user (modify button look with classes and display button label). Output is used for custom event which will be fired every time user click our button.

This component is small, focused, independent, testable and reusable. It does not contain any logic unrelated to behavior of the component itself. It has no idea of internals of rest of the application and can be safely imported and used in any part of the application or even in the other applications.

Refactored registration form component (container)

```
@Component({
  selector: 'app-registration-form',
  template: `<form [formGroup]="userService.form">
  <input type="text" [formControl]="userService.form.get('username')" placeholder="Ni
ckname">
  <input type="password" [formControl]="userService.form.get('password')" placeholder
="Password">
  <input type="password" [formControl]="userService.form.get('passwordConfirm')" plac
eholder="Confirm password">
  <app-button className="button accent" label="Register" (click)="registerUser()"></a
pp-button>
</form>
`
})
export class RegistrationFormComponent {
  constructor(public userService: UserService) {}

  registerUser() {
    const userFormData = this.userService.form.value;
    // some validation of user data
    // ...
    this.userService.registerUser(userFormData);
  }
}
```

You can see that our registration form now uses button and reacts on its click event with calling registerUser method. The logic of this method is tightly related to this form, so it is good place to put it here.

This was pretty simple example and we have only two levels in the component tree. **There are some pitfalls with this pattern when you have more levels in your component tree.**

More sophisticated example

This is not a real world example, but I hope it will help to understand possible issues with this pattern.

Imagine we have such component tree (from top to bottom):

user-orders - top-level component. It is container component which talks to services layer, receives data about user and their orders, passes it further the tree and renders the list of the orders.

user-orders-summary - middle level component. It is presentational component which renders the bar above the user orders list with total number of orders.

cashback - bottom level (leaf) component. It is presentational component which displays the total amount of user's

cashback and has a button to withdraw it to bank account.

Top-level container component

Let's look on our top-level user-orders container component.

```
@Component({
  selector: 'user-orders',
  templateUrl: './user-orders.component.html'
})
export class UserOrdersComponent implements OnInit {
  user$: Observable<User>;
  orders$: Observable<Order[]>;

  constructor(
    private ordersService: OrdersService,
    private userService: UserService
  ) {}

  ngOnInit() {
      this.user$ = this.userService.user$;
      this.orders$ = this.ordersService.getUserOrders();
  }

onRequestCashbackWithdrawal() {
    this.ordersService.requestCashbackWithdrawal()
      .subscribe(() => /* notification to user that cashback withdrawal has been requ
ested */);
    }
}
```

```
<div class="orders-container">
    <user-orders-summary
        [orders]="orders$ | async"
        [cashbackBalanace]="(user$  | async).cashBackBalance"
        (requestCashbackWithdrawal)="onRequestCashbackWithdrawal($event)"
    >
    </user-orders-summary>

    <div class="orders-list">
      <div class="order" *ngFor="let order of (orders$ | async)"></div>
    </div>
</div>
```

As you can see, user-orders component defines two observables: user$ and orders$, using async pipe in template to subscribe to them. It passes the data to presentational component user-orders-summary and also renders a list of orders. Also it talks to service layer reacting on custom event requestCashbackWithdrawal emitted from user-orders-summary.

Middle level presentational component

```
@Component({
  selector: 'user-orders-summary',
  template: `
```

```
    <div class="total-orders">Total orders: {{orders?.length}}</div>
    <cashback [balance]="cashbackBalanace" (requestCashbackWithdrawal)="onRequestCash
backWithdrawal($event)"></cashback>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserOrdersSummaryComponent {
    @Input() orders: Order[];
    @Input() cashbackBalanace: string;

    @Output() requestCashbackWithdrawal = new EventEmitter();

    onRequestCashbackWithdrawal() {
        this.requestCashbackWithdrawal.emit();
    }
}
```

This component is designed very similarly to our refactored button component. It renders the data received through its inputs and emits custom event on some user action. It does not call any services and does not contain any business logic. So this is the pure presentational component which uses another presentational cashback component.

Bottom level presentational component

```
@Component({
    selector: 'cashback',
    template: `
<div class="cashback">
  <span class="balance">Your cashback balance: {{balance}}</span>
  <button class="button button-
primary" (click)="onRequestCashbackWithdrawal()">Withdraw to Bank Account</button>
</div>
`,
    styleUrls: ['./cashback.component.css']
})
export class CashackComponent {
    @Input() balance: string;

    @Output() requestCashbackWithdrawal = new EventEmitter();

    onRequestCashbackWithdrawal() {
        this.requestCashbackWithdrawal.emit();
    }
}
```

This is another presentational component which only receives the data through input and throw events with output. Pretty simple and reusable, **but we have some problems in our component tree**.

You probably noticed that user-orders-summary component and cashback component have similar Inputs (cashbackBalanace and balance) and same Output (requestCashbackWithdrawal). That's because our container component is too far from deepest presentational component. And the more tree levels we will have with this design, the worse the problem will become. Let's look at this problems closer.

Issue 1 - Extraneous properties in middle level presentational components

Our user-orders-summary receives cashbackBalanace Input just to pass it further down the tree but not even use it by itself. If you face such situation, this is one of the markers that you probably may have component tree design flaw. Real life components may have a lot of inputs and outputs, and with this design you will have a lot of "proxy" inputs which will make your middle level components less reusable (as you tight them to children components) and a lot of repeatable code.

Issue 2 - Custom event bubbling from down to top-level components

This issue is very similar to previous one, but related to component's outputs. As you can see, requestCashbackWithdrawal custom event is repeated in cashback and user-orders-summary components. That again happens because our container component is too far in the tree from the deepest presentational component. And it also makes our middle component non-reusable by itself.

There are at least two possible solutions to this problems.

1st – make your middle level components more content agnostic using ngTemplateOutlet and expose your deeper components to container components directly. We will pass this for today as it deserves separate article.

2nd – redesign your component tree.

Refactoring our component tree

Let's refactor our code to see how we can solve the problems with extraneous properties and event bubbling in middle level component.

Refactored top-level component

```
@Component({
  selector: 'user-orders',
  templateUrl: './user-orders.component.html'
})
export class UserOrdersComponent implements OnInit {
  orders$: Observable<Order[]>;

  constructor(
    private ordersService: OrdersService,
  ) {}

  ngOnInit() {
      this.orders$ = this.ordersService.getUserOrders();
  }
}
```

```
<div class="orders-container">
    <user-orders-summary [orders]="orders$ | async"></user-orders-summary>

    <div class="orders-list">
      <div class="order" *ngFor="let order of (orders$ | async)"></div>
    </div>
</div>
```

We removed the user$ observable and onRequestCashbackWithdrawal() method from our top-level container

component. It is much simpler now and passes only the data needed to render user-orders-summary component itself, but not its child cashback component.

Middle level refactored component

```
@Component({
  selector: 'user-orders-summary',
  template: `
    <div class="total-orders">Total orders: {{orders?.length}}</div>
    <cashback></cashback>
    `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserOrdersSummaryComponent {
    @Input() orders: Order[];
}
```

It is also simplified a lot. Now it only has one Input and renders the total number of orders.

Bottom level refactored component

```
@Component({
    selector: 'cashback',
    template: `
<div class="cashback">
  <span class="balance">Your cashback balance: {{ (user$ | async).cashbackBalance }}<
/span>
  <button class="button button-
primary" (click)="onRequestCashbackWithdrawal()">Withdraw to Bank Account</button>
</div>
`,
    styleUrls: ['./cashback.component.css']
})
export class CashackComponent implements OnInit {
  user$: Observable<User>;

  constructor(
     private ordersService: OrdersService,
     private userService: UserService
  ) {}

  ngOnInit() {
    this.user$ = this.userService.user$;
  }

  onRequestCashbackWithdrawal() {
    this.ordersService.requestCashbackWithdrawal()
      .subscribe(() => /* notification to user that cashback withdrawal has been requ
ested */);
    }
  }
}
```

Wow. As you can see, **it is not presentational anymore**. Now it looks very similar to our top-level component, so it is container component down the component tree. This refactoring allowed us to simplify the whole component tree

design and APIs and logic of our two components up the component tree.

What about reusability of our new cashback component? It is still reusable, as it contains only the logic related to the component itself, so it is still independent.

The new design of our component tree seems to be more maintainable, more streamlined and more atomized. We have no bubbling events now, we have no repeatable inputs through the component tree and overall design is much simpler. We achieved this by putting additional container component down the component tree. This method can be used to simplify your component tree design, but you should have good understanding which components in your tree are suitable for being containers without big loss in reusability, and which should be pure presentational components. That's always a subject of balance and design choices when you create your app's architecture.

It's very easy to misunderstand the Container-Presentation pattern and think that container components can only be top-level components (intuitively, they contain all the other components in the local component tree). But this is not the case, container components can be on any level of the component tree, and as you saw, even on the leaf level. I like to call them **smart** components because for me it's much clear that these components will contain some business logic and can be presented anywhere in the component tree.

Afterword

I hope at this moment you have better overview on Container-Presentation pattern and possible issues with its implementation.

I tried to keep it as simple as possible, but there is a ton of information available related to this pattern.

If you have any questions or notes, feel free to reach me in the comments.

The next article will be about changeDetectionStrategy in Angular (which is highly related to this post).

See you!

#Angular #Angular2 #Coding Guidelines #Frontend #UI Development #Other